CS 115 Lecture 16

List algorithms, parallel lists

Neil Moore

Department of Computer Science
University of Kentucky
Lexington, Kentucky 40506
neil@cs.uky.edu

12 Nov 2015

Functions that mutate lists

Let's write a function that mutates a list.

- **Scaling**: multiply all the elements by the same number.
- Parameters: a list and a scaling factor.
- Postconditions: mutates the list and returns nothing.
- Usually a mutating function needs to loop over indices, not elements.
- `scale.py`
- What happens if we pass it a string instead of a list?

List algorithms

Let's look at and implement several algorithms for lists.

- Pretty much all list algorithms use a loop.
  - Usually a for loop, occasionally a while.
- Sum: add together all the elements.
- Count: find the number of occurrences of a value.
- Max/min: find the largest/smallest value.
- Sort: rearrange the elements to be in order.
- All of these are available as built-in functions or methods.
  - But we'll still write them ourselves. Why?
  - It's good to understand how they work.
  - And sometimes we need a slightly different variant.

Sum

Adding up the elements of a list works like adding up user input, which we've done before.

- Need an accumulator. What initial value?
  - 0 — the additive identity (adding 0 doesn't change anything)
- The algorithm:
  1. Initialize the accumulator to 0.
  2. For each element of the list, add it to the accumulator.
  3. Return the accumulator.
- In Python we can also use the built-in function `sum`:

```
total = sum(mylist)
```

- Variations: sum of squares, product, concatenation.
- `addup.py`

Count

The `in` operator tells us *whether* an value is in a list. Sometimes we also want to know *how many times* it is there.

- Two parameters: a list, and the value to search for.
- We'll need an accumulator again to keep track of the count.
  - In particular, a **counter**.
- The algorithm:
  1. Initialize the counter to 0.
  2. For each element of the list:
     (2.1) If it equals the search value, add one to the counter.
  3. Return the counter.
- Python lists have a `count` method:

```
numzeros = mylist.count(0)
```

- Variations: count the elements with a particular property.
- `count.py`

Maximum and minimum

What if we want to find the largest element?

- Use a variable to track the largest so far.
  - What to initialize it to?
  - 0? What if the list is all negative?
  - -999999? Same problem: the elements might all be smaller.
  - Use the first element of the list!
    - ★ "The largest" doesn't make sense on an empty list: error.
- The algorithm:
  1. Initialize the "best" variable to the first element.
  2. For each element in the rest of the list:

     (2.1) If it's bigger than the best, it is the new best.
  3. Return the best.
- Python has functions `max` and `min`:
  ```
  largest = max(mylist)
  ```
  - Elements must be comparable (all str or all numbers, not a mix)
- Variations: index of the maximum, maximum function value.
- `maximum.py`

Sorting

- We already know the `sort` function.
- But how does it work?
- There are several algorithms for sorting:
  - Selection sort, insertion sort, quick sort, merge sort.
  - `http://www.sorting-algorithms.com/`
  - Most of these algorithms are based around:
    - ★ Comparing elements.
    - ★ Then swapping them into the right place.
  - Different algorithms have different trade-offs:
    - ★ Some require fewer comparisons.
    - ★ Some require fewer swaps.
    - ★ Some require less memory.
    - ★ Some are good on "almost-sorted" data.
- We'll look at one algorithm: selection sort.
  - Not the fastest, but one of the simplest.
  - Also requires the fewest swaps.

Selection sort

The idea behind selection sort: iterate through the list in multiple passes:

- First, put the smallest element into the right place.
  - Can find the smallest with `min` and `index`.
  - Then swap it with the first element.
    `lst[0], lst[minpos] = lst[minpos], lst[0]`
  - This is pass 1.
- Then put the second-smallest element into the right place.
  - Use `min` and `index` on the unsorted part of the list.
  - Then swap it with the second element.
  - That's pass 2.
- And the third-smallest, and the fourth, and...
- Sounds like we need a loop!

Selection sort algorithm

- For each index in the list (each pass):
  1. Find the smallest element after index i.
  2. Swap that element with index i.
     Now all the elements up to index i are sorted.
- That's all!
  - Each pass makes more of the list sorted than before.
  - Gets us closer to the goal, but not all the way there.
  - Then repeat until we reach the goal: common algorithmic technique.
  - Have to make sure you're getting closer to the goal: in each pass, there are fewer numbers to sort than in the previous.
- It turns out we could stop before the last index. Why?
  - If everything else is in the right place, it must be too!
- `selsort.py`

Parallel lists

- Sometimes we need to store collections of related information:
  - Employees and salaries.
  - Songs, performers, and albums.
  - Monster locations and hit points.
- We can do this using multiple lists with matching indices.
  - So `songs[0]` goes with `artists[0]`, etc.
  - That means all the lists must be the same length.
  - These are called **parallel lists**.
- Python has other ways to do similar things:
  - Lists of lists, dictionaries, user-defined objects...
  - Parallel lists are the easiest to get started with.

Parallel list examples

- Suppose we have two parallel lists, of student names and scores.
- If I give you a name, how would you find their score?
  - Find the index of that name in the name list.
  - The score is at the same index in the other list.
- What if I wanted a list of all the students with "A"s?
  1. Build an accumulator list for the answer.
  2. Iterate over the score list (keeping track of the index)
     - (2.1) If the score is ≥ 90:
       - (2.1.1) Find the name at the same index.
       - (2.1.2) Append that name to the accumulator.
  3. Return the accumulator.
- Let's implement functions for both of these.
- `parallel.py`

Another example

- Another example related to grading. . . multiple-choice
- Let's say we have a list of the correct answers.
- . . . and we also have someone's answers to the same questions.
  - These are parallel lists!
- How can we calculate their score (number of right answers)?
  1. Keep an counter of the number of correct answers.
  2. For each index in the lists:
     (2.1) If the student's answer equals the correct answer:
     (2.1.1) Increment the counter.
  3. Return the counter.
- `gradequiz.py`

Something to think about

How would you sort parallel lists?

- Can you use the built-in `sort` method?
- No—because that sorts only one list.
- Can't we just call sort twice, once on each list?
  - No—that would scramble the associations.
  - Sorry, Aaron, you now have the lowest grade in class.
- We need one function that takes *two* lists.
  - Use selection sort, comparing the elements of one list.
  - But when you swap, swap the same positions in both lists.